Using Magnetic Disk Instead of Main Memory in the Mur φ Verifier

Ulrich Stern and David L. Dill

Computer Science Department, Stanford University,
Stanford, CA 94305
{uli, dill}@cs.stanford.edu

Abstract. In verification by explicit state enumeration a randomly accessed state table is maintained. In practice, the total main memory available for this state table is a major limiting factor in verification. We describe a version of the explicit state enumeration verifier Murφ that allows the use of magnetic disk instead of main memory for storing almost all of the state table. The algorithm avoids costly random accesses to disk and amortizes the cost of linearly reading the state table from disk over all states in a given breadth-first level. The remaining runtime overhead for accessing the disk is greatly reduced by combining the scheme with hash compaction. We show how to do this combination efficiently and analyze the resulting algorithm. In experiments with three complex cache coherence protocols, the new algorithm achieves memory savings factors of one to two orders of magnitude with a runtime overhead of typically only around 15%.

1 Introduction

Modern digital systems often have components that run concurrently. Interactions among these components are a notorious source of design errors. Conventional verification methods based on hand-generated test vectors and pseudo-random simulation are not capable in practice of finding all of these problems. Programs that exhaustively enumerate all reachable states of a part of the system (or an abstraction of the system), however, have been shown to be very effective at detecting bugs that are missed by other means. The reachability analysis in these formal verification tools can be performed using two different methods: the states can be explicitly enumerated, by storing them individually in a table, or a symbolic method can be used, such as representing the reachable state space with a binary decision diagram (BDD) [1].

In many applications, such as directory-based cache coherence protocols, BDD-based reachability analysis exhibits close to worst-case behavior. In such situations, reasonably efficient explicit enumeration can save a factor of 50 or more in space, because the size of the state table is the product of the number of reachable states and the number of bits to represent each state, while a BDD requires almost one node per bit per reachable state, and each node is approximately 20 bytes.

Recently, several techniques have been developed that allow significantly more complex systems to be handled using explicit state enumeration, especially when the techniques are used in combination. These techniques follow two different approaches. First, state reduction methods have been developed that aim at reducing the size of the reachability graph while ensuring that system errors will still be detected. Examples are exploiting symmetries, utilizing reversible rules, and employing repetition constructors [8], as well as partial order techniques [11]. These methods directly tackle the main problem in reachability analysis: the very large number of reachable states of most systems. The second approach aims at exploring a given reachability graph in the most efficient manner, minimizing memory usage and runtime (both of which are limiting factors in verification). Examples are bitstate hashing [5], hash compaction [17, 14], and parallelizing the state space search [16].

In this paper, we describe a technique that reduces the main memory requirements of the state table maintained in explicit state enumeration. The state table eventually holds all reachable states of the system being verified unless an error is detected. In addition, the state table is typically randomly accessed, in which case the use of magnetic disk for this table incurs a huge runtime penalty and hence main memory is required to store this table. In practice, the total main memory available for the state table is a major limiting factor in verification.

We describe a version of the explicit state verifier Murφ [4] that allows the use of magnetic disk instead of main memory for storing almost all of the state table, at the cost of only a small runtime overhead. The algorithm is based on the observation that when a breadth-first search is used to enumerate the state space, a newly generated state does not need to be checked against the state table immediately; in fact, one can postpone the checking until an entire level of the breadth-first search has been explored and then check all states in that level together by *linearly* reading the table from disk. This scheme avoids costly random accesses to disk and amortizes the time for accessing the full table on disk over all states in a given search level.

The remaining runtime overhead for accessing the disk can be greatly reduced by combining the new scheme with hash compaction. Hash compaction stores only hash signatures instead of full state descriptors in the state table. The resulting memory savings of typically two orders of magnitude and the resulting reduced disk access times come at a certain price; there is now a small probability that the verifier misses the error states of the system and incorrectly claims that an erroneous system is correct (i.e., produces a false positive). We derive an upper bound on this probability in the combined scheme and show that, e.g., 6-byte signatures are typically sufficient to reduce the bound to 0.1%.

One might be concerned about the reliability of a "probabilistic verifier" that can miss errors with a small probability. For several reasons, however, this concern is unjustified. First, the probability of missing an error due to hash compaction should not be confused with the probability of the very same error not occurring when running or simulating the system for a long time. The former probability is guaranteed to be very small even in situations where the latter is

high. Second, it is typically necessary to scale down or simplify a system of industrial size to make it amenable to formal verification, which also results in some probability of missed errors. In comparison to this probability, which cannot even be approximated, the probability of missed errors due to hash compaction seems negligible. Third, when re-running a probabilistic verifier with independent hash functions, the resulting probability of missed errors is the product of the probabilities in the two runs, which allows making the probability of missed errors arbitrarily small. For a more detailed explanation of why it is safe to use a probabilistic verifier see Sect. 1.3.1. in [13].

We ran experiments using the new scheme on three complex cache coherence protocols (SCI [7], DASH [10], and FLASH [9]), varying the ratio of the number of states stored on disk to the states in main memory. We call this ratio the *memory savings factor*. (The additional memory savings due to hash compaction are not taken into account here.) For example, with a memory savings factor of 50, the new scheme slowed down verification by an average of only 20% on an SGI Indy and an average of only 29% on a Sun UltraSPARC. In fact, the algorithm is shown to work well if the reachability graph of the system under verification has a small diameter, which is true for virtually all systems that have been studied with Murφ.

The algorithm presented in this paper was inspired by a scheme devised by Roscoe that allows the use of magnetic disk in explicit state enumeration [12]. His scheme seems more complicated than ours since it is based on an algorithm for sorting without randomly accessing memory. Also, one can show with a simple analysis that his scheme would induce a high runtime overhead. (He has not reported any empirical data about his scheme.) In addition, the file merging used in his scheme doubles the memory requirements of the state table on disk. A detailed comparison of the two algorithms is given in [13].

This paper is organized as follows. Section 2 provides background on explicit state enumeration and magnetic disk speed. The new algorithm that enables the use of magnetic disk instead of main memory for storing almost all of the state table is described and analyzed in Sect. 3. Results running the algorithm are reported in Sect. 4. Finally, Sect. 5 gives some concluding remarks.

2 Background

2.1 Explicit State Enumeration

In explicit state enumeration, the automatic verifier tries to examine all reachable states from a set of possible start states. Either breadth-first or depth-first search can be employed for the state enumeration process. Both the breadth-first and the depth-first algorithms are straightforward.

Two data structures are needed for performing the state enumeration. First, a state table stores all the states that have been examined so far and is used to decide whether a newly-reached state is old (has been visited before) or new (has not been visited before). Besides the state table, a state queue holds all

active states (states whose successors still need to be generated). Depending on the organization of this queue, the verifier does a breadth-first or a depth-first search.

2.2 Magnetic Disk Speed

The speed of a magnetic disk depends strongly on the way it is accessed. When linearly accessing a large file on disk, we have measured a read transfer rate of typically 3 MB/s and a write transfer rate of typically 1.5–2 MB/s. The seek time for a random access, however, is typically 10 ms. Thus, to read, say, a single word randomly from disk requires almost four orders of magnitude more time than to read one in the course of a linear access.

3 Explicit State Enumeration Using Magnetic Disk

3.1 The Basic Algorithm

The basic algorithm for explicit state enumeration using magnetic disk is given in Figure 1 and is described in the following paragraph. Note that the algorithm maintains two state tables: one in main memory and one on disk. The state queue and the disk table will be accessed only sequentially; the main memory table will be accessed randomly.

The state enumeration is started by calling SEARCH(). First, the startstates are generated and inserted into the main memory table by calling INSERT(). The search loop generates the successors for all states in the state queue and also inserts these successors into the main memory table. When the state queue becomes empty, the algorithm calls CHECKTABLE(), which determines those states in the main memory table that are new and inserts them into the state queue. Note that CHECKTABLE() linearly reads the disk table to sort out old states, and eventually clears the main memory table. Further note that if there is sufficient space in the main memory table, the algorithm will call CHECKTABLE() exactly once for each breadth-first level of the search; otherwise, if the main memory table fills up (because some breadth-first levels have too many states), CHECKTABLE() will also be called from within the INSERT() routine.

3.2 Estimating the Overhead

We now estimate the runtime overhead incurred by accessing the magnetic disk in our algorithm. Let k_i denote the number of states in the disk table when it is read for the ith time and assume that it is read a total of t times during the state space search. Note that $k_1 = 0$ since the disk table is empty the first time it is read. The total number of states read from disk is $\sum_{i=1}^{t} k_i$. This sum has its smallest possible value if the main memory table never fills up completely, as in this case the disk table is read exactly once for each breadth-first level (plus once for the successors of the states in the last level). In this case, $t = d + 2$,

```
var     // global variables
  M: hash table;      // main memory table
  D: file;            // disk table
  Q: FIFO queue;      // state queue

SEARCH()     // main routine
begin
  M := ∅; D := ∅; Q := ∅;      // initialization
  for each startstate s0 do     // startstate generation
    INSERT(s0);
  end
  do                            // search loop
    while Q ≠ ∅ do
      s := dequeue(Q);
      for all s' ∈ successors(s) do
        INSERT(s');
      end
    end
    CHECKTABLE();
  while Q ≠ ∅;
end

INSERT(s: state)     // insert state s in main memory table
begin
  if s ∉ M then begin
    insert s in M;
    if full(M) then
      CHECKTABLE();
  end
end

CHECKTABLE()     // do old/new check for main memory table
begin
  for all s ∈ D do     // remove old states from main memory table
    if s ∈ M then
      M := M − {s};
  end
  for all s ∈ M do     // handle remaining (new) states
    insert s in Q;
    append s to D;
    M := M − {s};
  end
end
```

Fig. 1. Explicit State Enumeration Using Magnetic Disk

where d denotes the diameter of the reachability graph. Since this diameter is typically quite small, the disk table will only be read a small number of times. (We shall see the diameters of some complex example protocols in Sect. 4.)

In an instance of the SCI protocol, for example, the (minimum) total number of states read from disk is $2.64 \cdot 10^7$. With 124 bytes per state and a disk bulk transfer rate of 3 MB/s, this would result in a runtime overhead of at least 1091 s. Comparing this value to the verification time (723 s) of the conventional algorithm on, for example, an UltraSPARC, yields a runtime overhead of at least 151 %. This overhead, however, can be reduced by combining the new algorithm with hash compaction.

Note that if a conventional verifier with randomly accessed state table runs out of main memory and is forced to do swapping, each checking of a newly generated state might require a seek. In fact, if the state table is much larger than the available main memory, we can assume that each checking does require a seek. For the above instance of the SCI protocol, $2.97 \cdot 10^6$ such seeks would be performed, resulting in a runtime overhead of $2.97 \cdot 10^4$ s, or 4108 %, assuming 10 ms per seek and verification on the above UltraSPARC.

3.3 Combining with Hash Compaction

Hash compaction reduces the memory requirements of the state table by storing (only) hash signatures instead of full state descriptors in this table. The resulting memory savings come at the price of a small probability, say, 0.1%, that the verifier incorrectly claims that an erroneous system is correct. For complex verification problems, hash compaction has achieved memory reduction factors of two orders of magnitude. Note that by reducing the number of bytes stored per state, hash compaction also reduces the time to read the disk table.

Figure 2 shows the new INSERT() and CHECKTABLE() routines when using hash compaction. Note that signatures are used for both main memory table M and disk table D; full state descriptors, however, need to be stored in the state queue Q, since successors cannot be generated from a signature. In the INSERT() routine, first the signature is calculated from the state descriptor with a hash function. Then, state descriptor and signature are stored in the state queue, while only the signature is stored in the main memory table. The state queue will hold two types of states: unchecked states (i.e., states for which it has not yet been checked whether they are 'old' or 'new') and states that are known to be 'new.' The two types of states partition the state queue into two parts and thus an implementation need only store the position of the border between the two parts.

The CHECKTABLE() routine first deletes all 'old' states from the main memory table, and then checks for all unchecked states in the queue whether they are 'old' or 'new.' While the 'old' states are deleted from the queue, the 'new' ones are appended to the disk table. Note that the checking of the state queue can be done by linearly reading the unchecked part of the queue. When storing the signatures separately from the state descriptors in a second queue, the algorithm need only (linearly) read the unchecked part of this small second queue. This

```
INSERT(s: state)      // insert s in main memory table and state queue
begin
  h := hash(s);       // calculate signature
  if h ∉ M then begin
    insert h in M;
    insert (s, h) in Q;
    if full(M) then
      CHECKTABLE();
  end
end

CHECKTABLE()          // do old/new check for main memory table
begin
  for all h ∈ D do    // remove old states from main memory table
    if h ∈ M then
      M := M − {h};
  end
  for all unchecked (s, h) ∈ Q do   // remove old states from state queue
    if h ∈ M then                   // and add new states to disk table
      append h to D;
      M := M − {h};
    else
      Q := Q − {(s, h)};
  end
end
```

Fig. 2. INSERT() and CHECKTABLE() routines when using hash compaction

results in a significant improvement, because as shown in [13], the state queue can become quite large in practice.

The new algorithm has another nice property: states are inserted into the disk table in the order of their exploration. This property enables using the scheme proposed in [15] to store the information needed for error trace generation in a file, which contains for each reachable state a record with two elements – the state's signature and the position (in the file) of the record of the state's predecessor. Since the disk table already contains each state's signature, additional storage is only required for the values for the positions of each state's predecessor. These values can be stored in a separate file to avoid slowing down accesses to the disk table.

3.4 Analysis of the Combined Scheme

The following analysis yields an upper bound on the probability of false positives, i.e., on the probability that the verifier incorrectly claims that an erroneous system is correct. This probability will be denoted by p_{om}.

As in [15], one can show that

$$p_{\text{om}} \leq 1 - \prod_{i=2}^{t} p_{k_i - 1} \ ,$$

where p_k denotes the probability that there is no omission (identical signature) when inserting a new state into a state table with a total of k states in main memory and on disk, and k_i denotes the number of states in the disk table when it is read for the ith time. We assume that the hash function yields signatures distributed uniformly over $\{0, \ldots, l-1\}$. (Universal hashing [2], used in Murφ, can be shown to distribute at least as well as uniformly. In addition, by choosing the hash function at random when the verifier is started, universal hashing distributes well *independently* of the system under verification.) Thus, the probability p_k can be bounded as $p_k \geq 1 - k/l$. Hence,

$$p_{\text{om}} \leq 1 - \prod_{i=2}^{t} (1 - \frac{k_i - 1}{l}) \ . \tag{1}$$

This formula can be used by the verification tool to calculate (and report) a bound on the probability of false positives.

Next, we derive a formula for an approximate bound on the probability of false positives, in order to estimate the number of bits needed for the signatures. For p_{om} to become small, it has to hold that $\sum_{i=2}^{t} k_i \ll l$. Then, using $e^x \approx 1 + x$ for $\mid x \mid \ll 1$, one can approximate the right-hand side of (1) as $\sum_{i=2}^{t} (k_i - 1)/l$. Assuming linear growth of the disk table, i.e., $k_i \approx n\, i/t$, where n denotes the number of reachable states, and a moderately large t, an approximate bound $\tilde{P_{\text{om}}}$ on the probability of false positives can be derived, namely

$$\tilde{P_{\text{om}}} = \frac{n\, t}{2\, l} \ .$$

Table 1 gives values for $\tilde{P_{\text{om}}}$ assuming $n = 10^9$ reachable states while varying the number of bits b for the signatures ($l = 2^b$) and the number of times t the disk table is read. The diameters of the systems we examined were typically quite small (less than 100) and similarly were the numbers of times the disk table was read. Note that 6-byte signatures yield an approximate bound $\tilde{P_{\text{om}}}$ on the order of 0.1% for the chosen values of n and t.

In comparison to the main memory version of hash compaction [15], the disk version needs approximately two times the number of bits b for the signatures. This increase is due to the fact that in the disk version a newly reached state is compared against almost all of the states in the state table, while in the main memory version it is compared against only a few of the states in the table. Since the memory savings factor achievable with the new scheme is typically one or two orders of magnitude, however, the doubling of the size of the signatures amounts to an insignificant penalty.

Table 1. Approximate bounds $\tilde{P}_{\text{om}}$ on the probabilities of false positives for $n = 10^9$

b	t		
	200	500	1000
40	9.1%	23%	45%
48	0.036%	0.089%	0.18%

4 Results on Sample Protocols

Figures 3 and 4 show the measured slowdown of the new scheme on an Indy and on an UltraSPARC, for instances of the SCI, DASH, and FLASH protocols. Some parameters of these instances are shown in Table 2. The protocols were scaled to provide interesting data and yet prevent the process of running the examples from becoming too time-consuming. The slowdown graphs show that the main memory requirements of the Murφ verifier can be reduced by one or two orders of magnitude with only a small increase in runtime.

Table 2. Example protocols

protocol	reachable states	bytes/ state	diameter	conventional scheme's runtime	
				Indy	UltraSPARC
SCI	1179 942	124	46	1437s	723s
DASH	254 937	532	64	2429s	1287s
FLASH	1021 464	136	45	2739s	2500s

The slowdown for the new algorithm was calculated relative to the most recent release (3.0) of Murφ, which was optimized for running in main memory. In fact, the disk version of Murφ is based on this main memory version of Murφ, which also contains symmetry reduction (which was employed in the above experiments). We have only partially optimized the disk version; in particular, the code for the main memory table, which is used much more often in the disk version than in the main memory version, could probably be optimized.

For our slowdown measurements, we did not reduce the size of the main memory; instead, we reduced the size of the main memory table to yield the desired memory savings factors. There was usually main memory left for the Unix file system buffer cache, which had not been disabled, since it turned out to not be feasible to disable it. Thus, the measured slowdowns might actually be smaller than the slowdown in the case when the verifier is really running out of main memory. Estimating the minimum slowdown from the amount of data read from disk, however, shows that the measured slowdown is typically higher than this minimum slowdown. Hence, the effect of the buffer cache cannot have had a dominating impact on our measurements.

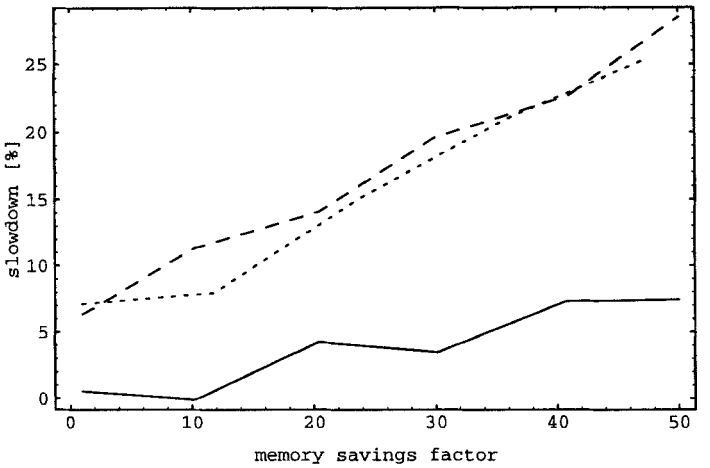

Fig. 3. Slowdown for the SCI (*dotted*), DASH (*solid*), and FLASH (*dashed*) protocols, calculated from the average runtime over three runs on an Indy

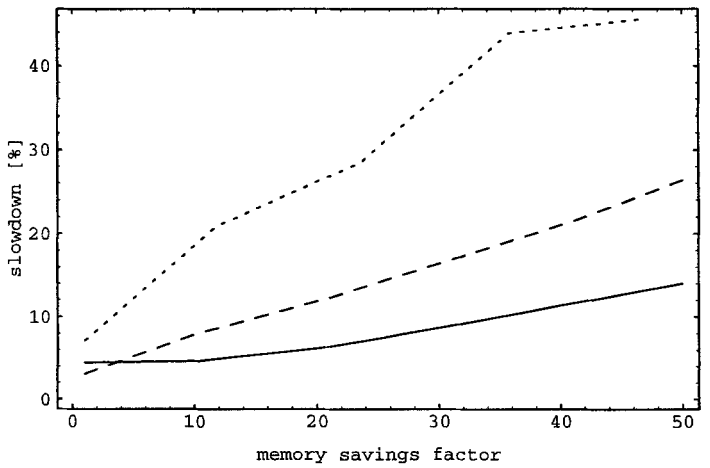

Fig. 4. Slowdown for the SCI (*dotted*), DASH (*solid*), and FLASH (*dashed*) protocols, calculated from the average runtime over eight runs on an UltraSPARC

5 Conclusions and Future Research

This paper describes a version of the explicit state enumeration verifier Murφ that allows the use of magnetic disk instead of main memory for storing almost all of the state table, at the cost of a small runtime overhead. The algorithm avoids slow random accesses to disk and amortizes the time for linearly reading the state table from disk over all states in a given breadth-first level. The remaining runtime overhead for accessing the disk is greatly reduced by combining the scheme with hash compaction. In experiments with three complex cache coherence protocols, the new algorithm achieved memory savings factors of one to two orders of magnitude with a runtime overhead of typically only around 15%. Hence, the algorithm can be used to tackle more complex problems or to run large verification jobs on a local workstation instead of a dedicated verification machine with a huge main memory.

The algorithm described could also be used in other explicit state verification tools like SPIN [6]. In addition, the algorithm is compatible with all three state reduction techniques in Murφ [8], with hash compaction, and with the parallel version of Murφ [16]. The algorithm is also compatible with Peled's partial order method [11], which had been assumed to require depth-first search, but was recently shown to also work with breadth-first search [3], on which the new scheme is based. This recent finding suggests that other partial order methods might also work with breadth-first search.

For checking liveness properties, all currently known efficient algorithms require a (modified) depth-first search of the state space. Hence, the new scheme is not directly compatible with these algorithms. Two other recent techniques that allow bigger state spaces, the most advanced version of hash compaction [15] and the parallel version of Murφ [16], however, also require a breadth-first search. Hence, checking liveness properties with a breadth-first style algorithm seems to be an interesting area for future research.

Acknowledgments

We would like to thank Ben Verghese for explaining some details of the Unix file system buffer cache to us and Ravi Soundararajan for his comments on a draft of this paper.

This work was supported in part by the Defense Advanced Research Projects Agency through NASA contract NAG-2-891 and a scholarship from the German Academic Exchange Service (DAAD-Doktorandenstipendium HSP-II). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, NASA, or the US Government.

References

1. J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *27th ACM/IEEE Design Automation Conference*, pages 46–51, 1990.
2. J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–54, 1979.
3. C.-T. Chou and D. Peled. Formal verification of a partial-order reduction technique for model checking. In *Tools and Algorithms for the Construction and Analysis of Systems. 2nd International Workshop*, pages 241–57, 1996.
4. D. L. Dill. The Murφ verification system. In *Computer Aided Verification. 8th International Conference*, pages 390–3, 1996.
5. G. J. Holzmann. On limits and possibilities of automated protocol analysis. In *Protocol Specification, Testing, and Verification. 7th International Conference*, pages 339–44, 1987.
6. G. J. Holzmann and D. Peled. The state of SPIN. In *Computer Aided Verification. 8th International Conference*, pages 385–9, 1996.
7. *IEEE Std 1596-1992, IEEE Standard for Scalable Coherent Interface (SCI).*
8. C. N. Ip. *State Reduction Methods for Automatic Formal Verification.* PhD thesis, Stanford University, 1996.
9. J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH multiprocessor. In *21st Annual International Symposium on Computer Architecture*, pages 302–13, 1994.
10. D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford DASH multiprocessor. *Computer*, 25(3):63–79, 1992.
11. D. Peled. Combining partial order reductions with on-the-fly model-checking. In *Computer Aided Verification. 6th International Conference*, pages 377–90, 1994.
12. A. W. Roscoe. Model-checking CSP. In *A Classical Mind, Essays in Honour of C. A. R. Hoare.* Prentice-Hall, 1994.
13. U. Stern. *Algorithmic Techniques in Verification by Explicit State Enumeration.* PhD thesis, Technical University of Munich, 1997.
14. U. Stern and D. L. Dill. Improved probabilistic verification by hash compaction. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 206–24, 1995.
15. U. Stern and D. L. Dill. A new scheme for memory-efficient probabilistic verification. In *Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification*, pages 333–48, 1996.
16. U. Stern and D. L. Dill. Parallelizing the Murφ verifier. In *Computer Aided Verification. 9th International Conference*, pages 256–67, 1997.
17. P. Wolper and D. Leroy. Reliable hashing without collision detection. In *Computer Aided Verification. 5th International Conference*, pages 59–70, 1993.